NVIDIA PAX-HPC Workshop 17th January 2024

Paul Graham | Senior Solutions Architect

pgraham@nvidia.com

Agenda

- 10:15-1045 Introduction to Heterogeneous Parallel Computing
- 10:45-11:00 Break
- 11:00-12:00 Key ways to accelerate applications
- 12:00-13:00 Programming for GPUs
- 13:00-13:45 Lunch
- 13:45-17:15 Hands-on practical

Some material …

- Open Hackathons Org: https://www.openhackathons.org/s/
    - GitHub Training Materials: https://github.com/openhackathons-org …
        - Including HPC & AI, profiling, multi-GPU
    - Plus, "n-ways to GPU programming": https://github.com/openhackathons-org/nways_accelerated_programming - a basis for the material in this talk

- DLI
    - GPU Acceleration with the C++ Standard Library
    - Scaling GPU-Accelerated Applications with the C++ Standard Library
    - Fundamentals of Accelerated Computing with OpenACC
    - An Even Easier Introduction to CUDA
    - Getting Started with Accelerated Computing with CUDA/C++
    - Accelerating CUDA C++ Applications with Concurrent Streams
    - Scaling Workloads Across Multiple GPUs with CUDA C++

Programming the NVIDIA Platform

ACCELERATED STANDARD LANGUAGES

ISO C++, ISO Fortran

```
std::transform(par, x, x+n, y, y,
    [=](float x, float y){ return y +
a*x; }
);
```

```
do concurrent (i = 1:n)
   y(i) = y(i) + a*x(i)
enddo
```

```
import cunumeric as np
…
def saxpy(a, x, y):
    y[:] += a*x
```

INCREMENTAL PORTABLE OPTIMIZATION

OpenACC, OpenMP

```
#pragma acc data copy(x,y) {
...
std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});
...
}
```

```
#pragma omp target data map(x,y) {
...
std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});
...
}
```

PLATFORM SPECIALIZATION

CUDA

```
__global__
void saxpy(int n, float a,
           float *x, float *y) {
  int i = blockIdx.x*blockDim.x +
          threadIdx.x;
  if (i < n) y[i] += a*x[i];
}

int main(void) {
  ...
  cudaMemcpy(d_x, x, ...);
  cudaMemcpy(d_y, y, ...);

  saxpy<<<(N+255)/256,256>>>(...);

  cudaMemcpy(y, d_y, ...);
```

ACCELERATION LIBRARIES

Core | Math | Communication | Data Analytics | AI | Quantum

Programming: ISO C++

Quick Background

C++ STL Containers

- One driving feature of C++ are its templates and the STL library. C++11 is further pushing these ideas and shows no sign of slowing.
- C++ templates are probably most widely used through the STL containers.
  - std::vector, std::string, std::map, std::list, etc...
- Besides the OO features and convenience, these containers are designed to rise-above basic C pointers, providing more safety from memory violations, while maintaining the bare-metal performance.
- For example std::vector → The vector template is designed to replace C's arrays.

```
std::vector<int> my_ints(4, 100);   // four ints with value 100
```

STD::PAR

What is std::par?

- Use standard C++ constructs to make code run parallel on heterogenous hardware
- C++11 introduced a memory model, concurrent execution model, and concurrency library, providing a standard way to take advantage of multicore processors
- The C++17 Standard introduced higher-level parallelism features that allow users to request parallelization of Standard Library algorithms.

Advantage:

- No language extensions, pragmas, directives, or non-standard libraries
- Write Standard C++, which is portable to other compilers and systems
- Compiler automatically accelerates code with high-performance NVIDIA GPUs and hence less time porting and more time on what really matters

STD::PAR

Parallelism in Standard C++

- Parallelism is expressed by adding an execution policy as the first parameter to any algorithm that supports execution policies
- Most of the existing Standard C++ algorithms were enhanced to support execution policies

Execution policies can be applied to most standard algorithms

- std::execution::seq = sequential: Sequential execution. No parallelism is allowed.
- std::execution::par = parallel: Parallel execution on one or more threads.
- std::execution::par_unseq = parallel + vectorized: Parallel execution on one or more threads, with each thread possibly vectorized.

C++17 PARALLEL ALGORITHMS

Example

```
C++98: std::sort(c.begin(), c.end());
```

```
C++17: std::sort(std::execution::par, c.begin(), c.end());
```

- NVC++ can compile Standard C++ algorithms with the parallel execution policies std::execution::par execution on NVIDIA GPUs.
- An NVC++ command-line option, -stdpar, is used to enable GPU-accelerated C++ Parallel Algorithms

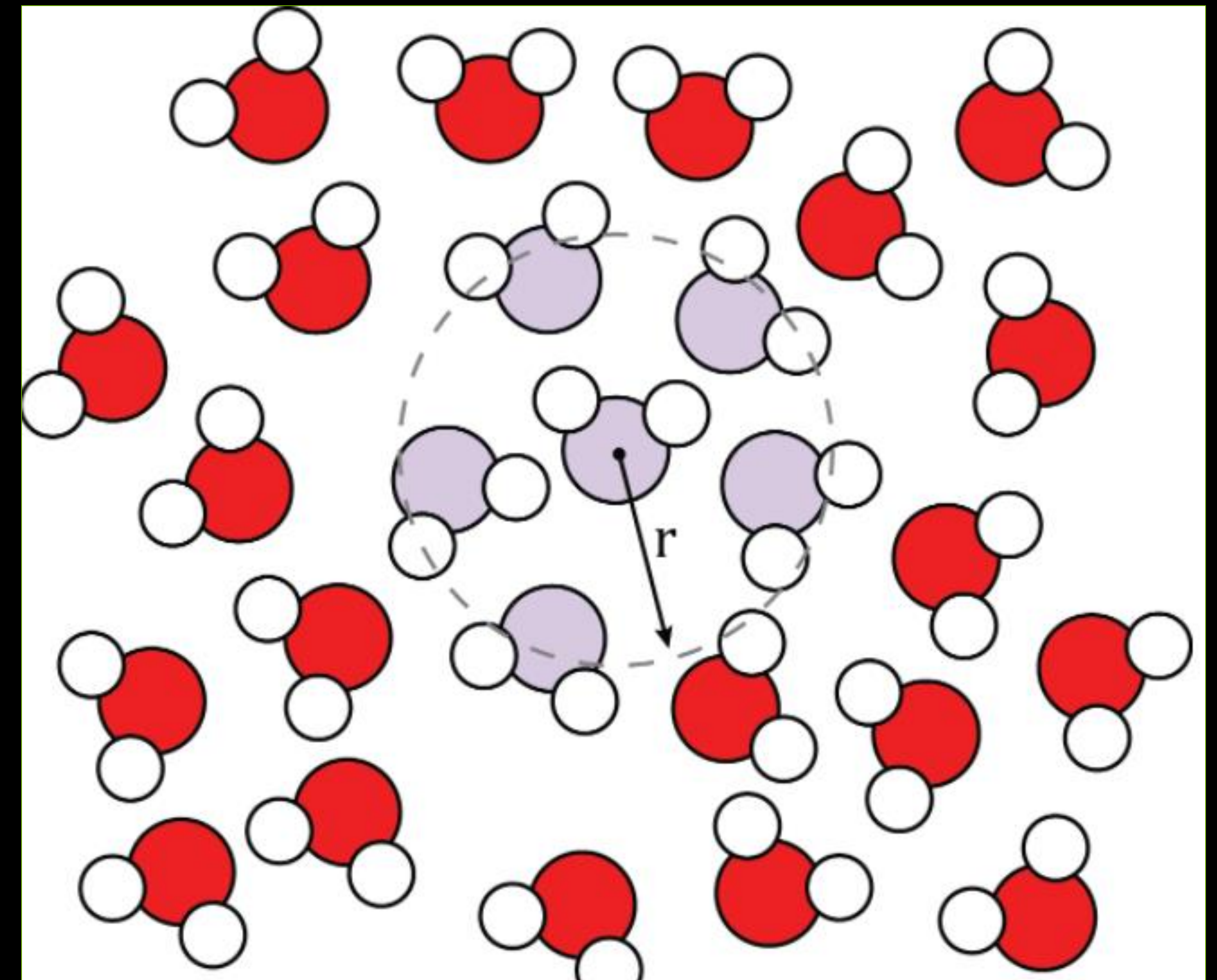
```
nvc++ -stdpar program.cpp -o program
```

Application

Molecular Simulation

RDF

The radial distribution function (RDF) denoted in equations by g(r) defines the probability of finding a particle at a distance *r* from another tagged particle.



Radial Distribution Function

Pseudo Code

```
for (int frame=0;frame<nconf;frame++){

    for(int id1=0;id1<numatm;id1++)
    {
        for(int id2=0;id2<numatm;id2++)
        {
            dx=d_x[]-d_x[];
            dy=d_y[]-d_y[];
            dz=d_z[]-d_z[];
            r=sqrtf(dx*dx+dy*dy+dz*dz);

            if (r<cut) {
                ig2=(int)(r/del);
                d_g2[ig2] = d_g2[ig2] +1
;
            }
        }
    }
}
```

- Across Frames
- Find Distance
- Reduction

NVIDIA

STEPS

Step 1: Replace for with std::for_each

```
std::for_each (InputIterator start_iter, InputIterator last_iter, Function fnc)
```

start_iter : The beginning position from where function operations are to be executed.

last_iter : The ending position till where function has to be executed.

fnc/obj_fnc : The 3rd argument is a function or an object function which operation would be applied to each element.

NVIDIA.

STEPS

Step 2: Pass execution policy as std::execution::par

```
for_each (std::execution::par , InputIterator first, InputIterator last, Function fn)
```

Execution policy as the first parameter:
execution:par will dictate that the loop body should execute in parallel across threads

NVIDIA.

STEPS

Step 3: Change indexing to use counting::iterator

```
std::for_each(std::execution::par,
    thrust::counting_iterator<unsigned int>(0u),
    thrust::counting_iterator<unsigned int>(numatm*numatm), ... )
```

```
std::vector<unsigned int> indices(numatm * numatm);
std::generate(indices.begin(), indices.end(), [n = 0]() mutable { return n++; });

std::for_each(std::execution::par,
    indices.begin(), indices.end(),
```

- Counting Iterator helps in filling up a vector with the numbers zero through N
- In our case from 0 to number of atoms squared
- GPU will be using the Thrust library for counting iterator for GPU
  - High-Level Parallel Algorithms Library
  - Parallel Analog of the C++ Standard Template Library (STL)

NVIDIA.

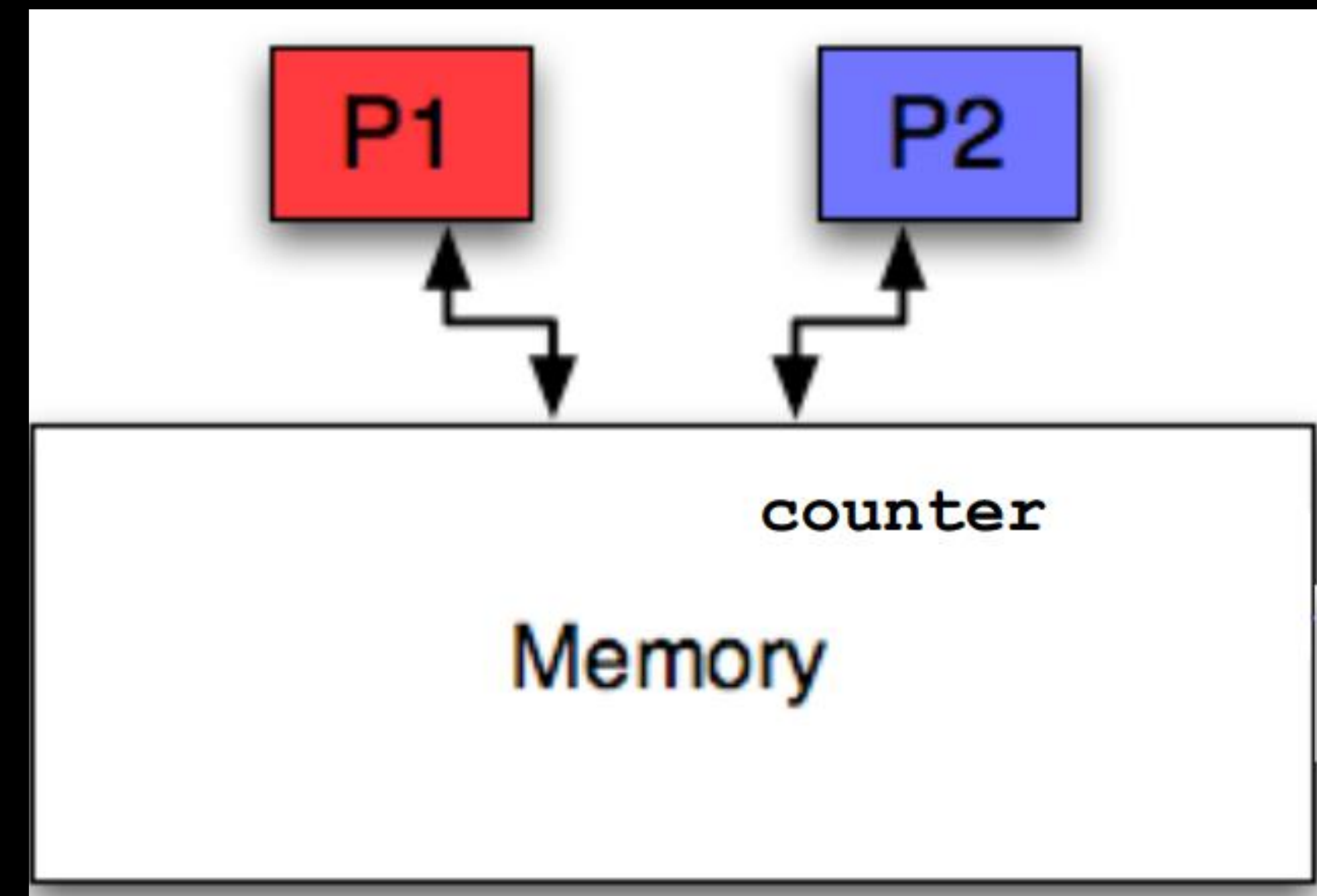
Atomic

Step 4: Remove Datarace

```
std::atomic<int>* d_g2 = new std::atomic<int>[nbin];
```

```
void *do_stuff(void * arg)
{
        for (int i = 0 ; i < 200000000 ; ++ i)
        {  counter ++; }
        return arg;
}
```

Since the variable counter is shared, we can get a data race



STEPS

Step 5: Put function body inside Lambda

```
std::for_each(std::execution::par,
    thrust::counting_iterator<unsigned int>(0u),
    thrust::counting_iterator<unsigned int>(numatm*numatm)
[...](unsigned int index)
        {
            for(int id2=0;id2<numatm;id2++)
            {
                dx = d_x[]-d_x[];
                dy = d_y[]-d_y[];
                dz = d_z[]-d_z[];
                r = sqrtf(dx*dx+dy*dy+dz*dz);

                if (r<cut) {
                    ig2=(int)(r/del);
                    ++d_g2[ig2];
                }
            }
        }
    )
```

- Lambda : Convenient way of defining an anonymous function

STEPS

Step 6: Compile for Multicore and GPU

```
std::atomic<int>* d_g2 = new std::atomic<int>[nbin];

std::for_each(std::execution::par,
        thrust::counting_iterator<unsigned int>(0u),
        thrust::counting_iterator<unsigned int>(numatm*numatm),
    [...](unsigned int index)
        {
            for(int id2=0;id2<numatm;id2++)
            {
                dx=d_x[]-d_x[];
                dy=d_y[]-d_y[];
                dz=d_z[]-d_z[];
                r=sqrtf(dx*dx+dy*dy+dz*dz);

                if (r<cut) {
                    ig2=(int)(r/del);
                    ++d_g2[ig2];
                }
            }
        }
    )
```

- Atomic Declaration
- Counting Iterator
- Find Distance
- Atomic Increment

```
nvc++ -stdpar=multicore program.cpp -o program
nvc++ -stdpar=gpu program.cpp -o program
```

NVIDIA.

Programming: ISO FORTRAN

Fortran

DO CONCURRENT :: ISO Standard Fortran

- ISO Standard Fortran 2008 introduced the DO CONCURRENT construct to allow you to express loop-level parallelism, one of the various mechanisms for expressing parallelism directly in the Fortran language
- HPC SDK 20.11 release of the NVIDIA HPC SDK, the included NVFORTRAN compiler automatically accelerates DO CONCURRENT

```
1  subroutine saxpy(x,y,n,a)
2    real :: a, x(:), y(:)
3    integer :: n, i
4    do i = 1, n
5      y(i) = a*x(i)+y(i)
6    enddo
7  end subroutine saxpy
```

```
1  subroutine saxpy(x,y,n,a)
2    real :: a, x(:), y(:)
3    integer :: n, i
4    do concurrent (i = 1: n)
5      y(i) = a*x(i)+y(i)
6    enddo
7  end subroutine saxp
```

```
nvfortran –stdpar=gpu,multicore program.f90 -o program
```

NVIDIA

ACCELERATED PROGRAMMING IN ISO FORTRAN

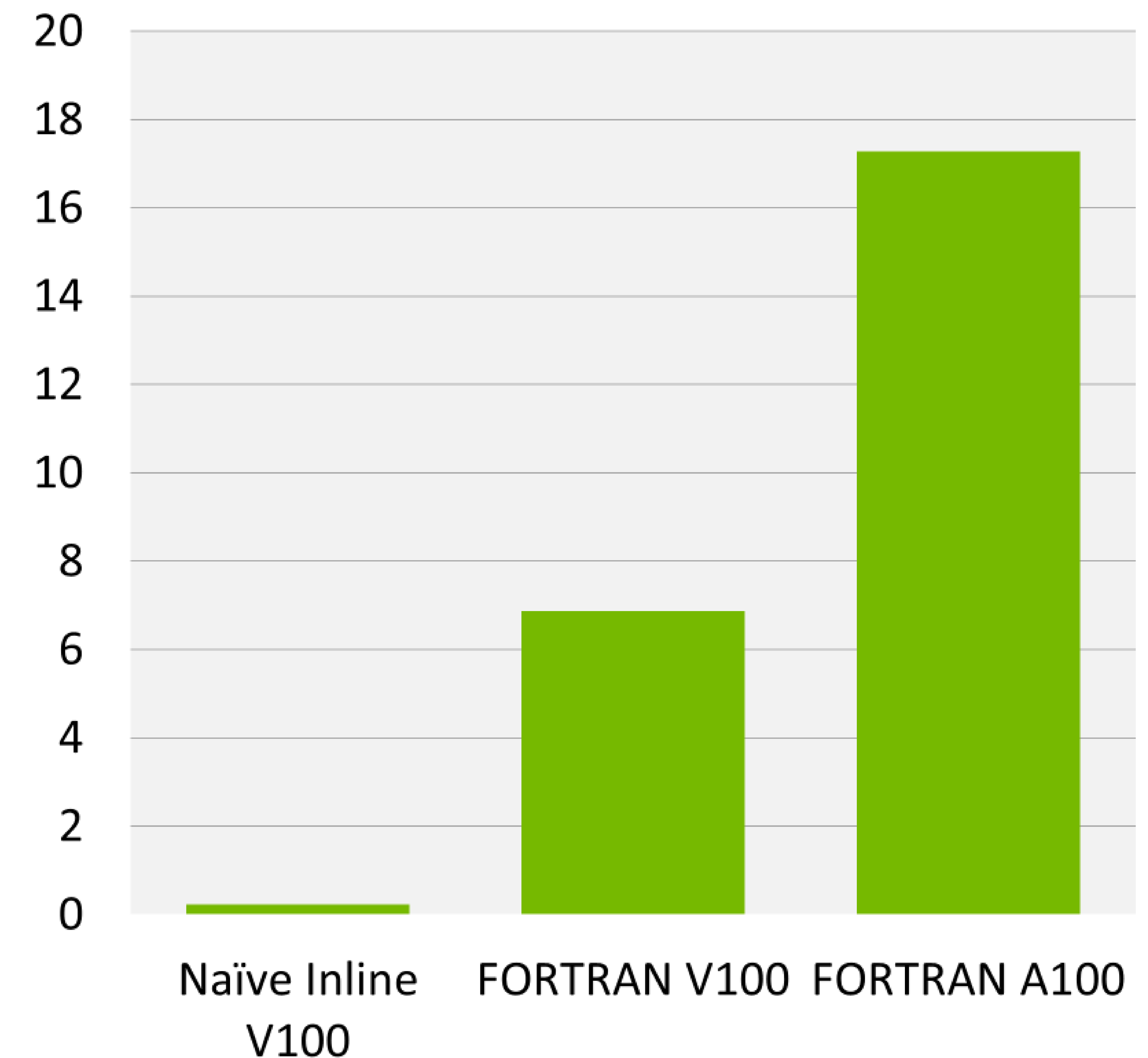
NVFORTRAN Accelerates Fortran Intrinsics with cuTENSOR Backend

```
real(8), dimension(ni,nk) :: a
real(8), dimension(nk,nj) :: b
real(8), dimension(ni,nj) :: c
...
!$acc enter data copyin(a,b,c) create(d)

do nt = 1, ntimes
  !$acc kernels
  do j = 1, nj
    do i = 1, ni
      d(i,j) = c(i,j)
      do k = 1, nk
        d(i,j) = d(i,j) + a(i,k) * b(k,j)
      end do
    end do
  end do
    !$acc end kernels
end do
!$acc exit data copyout(d)
```

Inline FP64 matrix multiply

```
real(8), dimension(ni,nk) :: a
real(8), dimension(nk,nj) :: b
real(8), dimension(ni,nj) :: c

...

do nt = 1, ntimes
  d = c + matmul(a,b)
end do
```

MATMUL FP64 matrix multiply



References

- **Accelerating Fortran DO CONCURRENT with GPUs and the NVIDIA HPC SDK (blog):** https://developer.nvidia.com/blog/accelerating-fortran-do-concurrent-with-gpus-and-the-nvidia-hpc-sdk/
- **Accelerating Standard C++ with GPUs Using stdpar (blog):** https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar/
- **C++ Standard Parallelism (webinar):** https://www.nvidia.com/en-us/on-demand/session/gtcspring22-s41960/
- **Developing Accelerated Code with Standard Language Parallelism (blog):** **https://developer.nvidia.com/blog/developing-accelerated-code-with-standard-language-parallelism/**
- **Multi-GPU Programming with Standard Parallel C++, Part 1 (blog)**: https://developer.nvidia.com/blog/multi-gpu-programming-with-standard-parallel-c-part-1/

Programming: Directives

OpenACC is...

a directives-based

parallel programming model

designed for

performance and **portability**.

Add Simple Compiler Directive

```
main()
 {
  <serial code>
  #pragma acc kernels
  {
   <parallel code>
  }
}
```

GAUSSIAN 16

Mike Frisch, Ph.D.
President and CEO
Gaussian, Inc.

"Using OpenACC allowed us to continue development of our fundamental algorithms and software capabilities simultaneously with the GPU-related work. In the end, we could use the same code base for SMP, cluster/ network and GPU parallelism. PGI's compilers were essential to the success of our efforts."

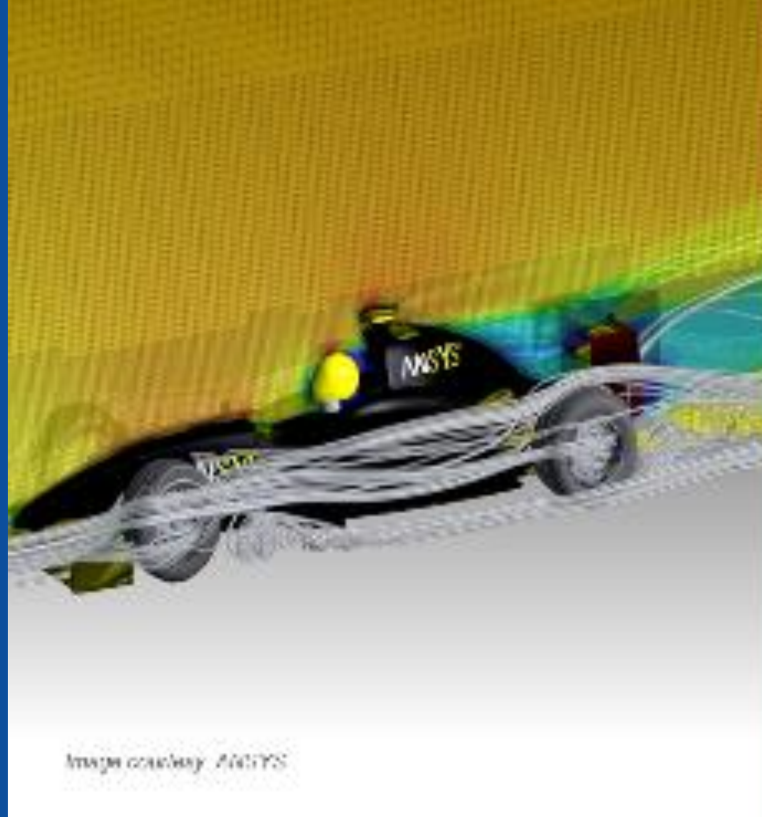
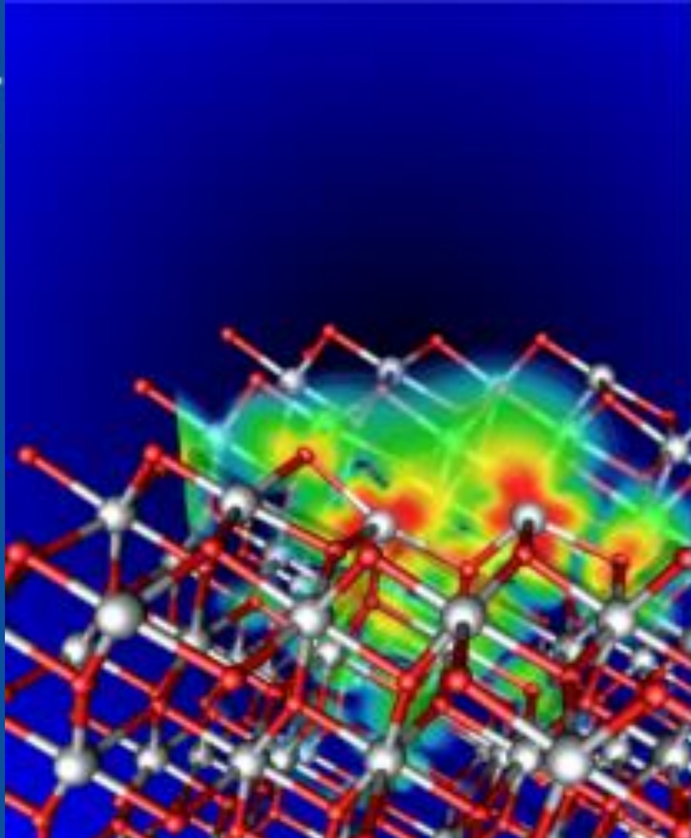


Image courtesy: ANSYS

ANSYS FLUENT

Sunil Sathe
Lead Software Developer
ANSYS Fluent

"We've effectively used OpenACC for heterogeneous computing in ANSYS Fluent with impressive performance. We're now applying this work to more of our models and new platforms."



VASP

Prof. Georg Kresse
Computational Materials Physics
University of Vienna

"For VASP, OpenACC is *the* way forward for GPU acceleration. Performance is similar and in some cases better than CUDA C, and OpenACC dramatically decreases GPU development and maintenance efforts. We're excited to collaborate with NVIDIA and PGI as an early adopter of CUDA Unified Memory."

COSMO

Dr. Oliver Fuhrer
Senior Scientist
Meteoswiss

"OpenACC made it practical to develop for GPU-based hardware while retaining a single source for almost all the COSMO physics code."

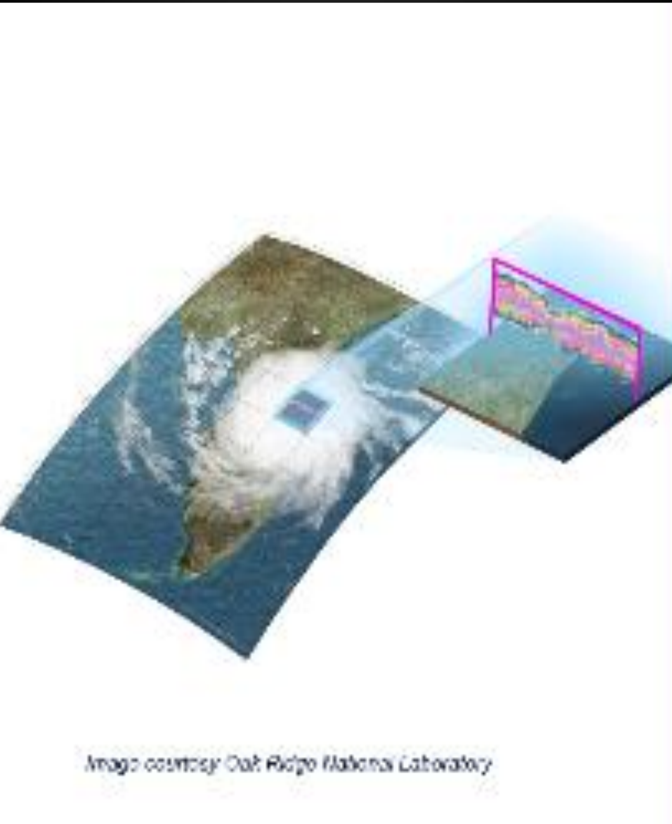


Image courtesy: Oak Ridge National Laboratory

E3SM

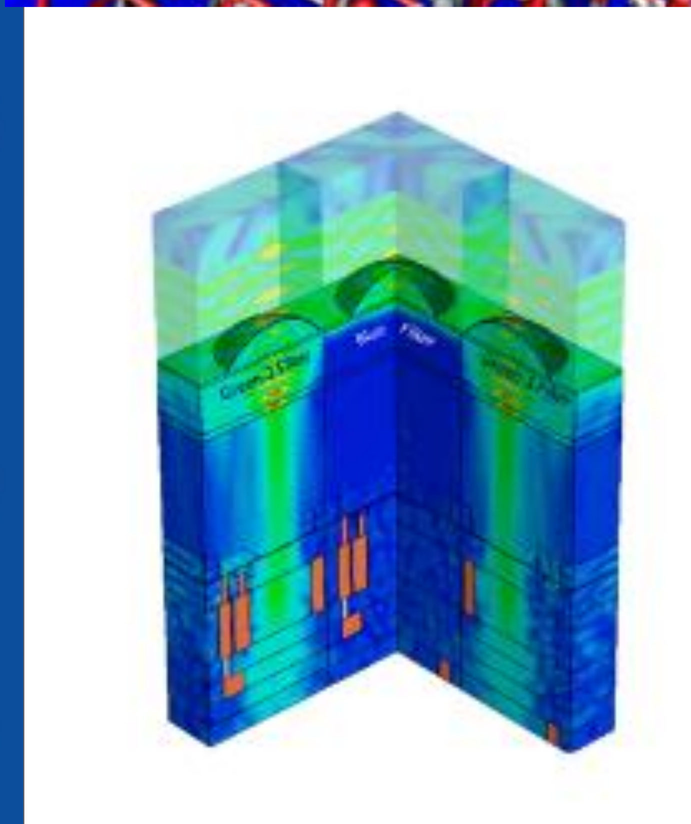
Mark A. Taylor
Multiphysics Applications
Sandia

"The CAAR project provided us with early access to Summit hardware and access to PGI compiler experts. Both of these were critical to our success. PGI's OpenACC support remains the best available and is competitive with much more intrusive programming model approaches."

NUMECA FINE/Open

David Gutzwiller
Lead Software Developer
NUMECA

"Porting our unstructured C++ CFD solver FINE/Open to GPUs using OpenACC would have been impossible two or three years ago, but OpenACC has developed enough that we're now getting some really good results."



SYNOPSYS

Dr. Lutz Schneider
Senior R&D Engineer
Synopsys Inc.

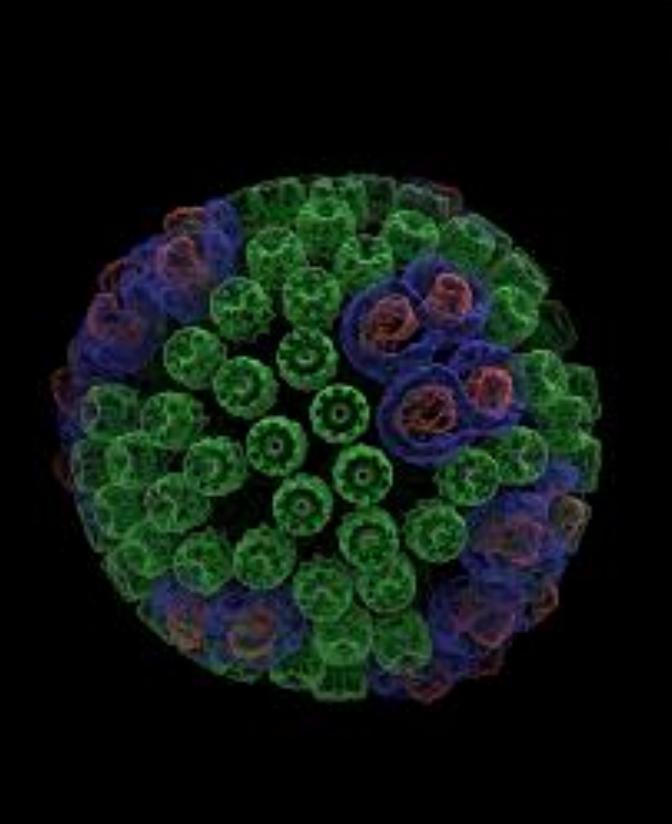
"Using OpenACC, we've GPU-accelerated the Synopsys TCAD Sentaurus Device EMW simulator to speed up optical simulations of image sensors. GPUs are key to improving simulation throughput in the design of advanced image sensors."

Image courtesy: NCAR

MPAS-A

Richard Loft
Director, Technology Development
NCAR

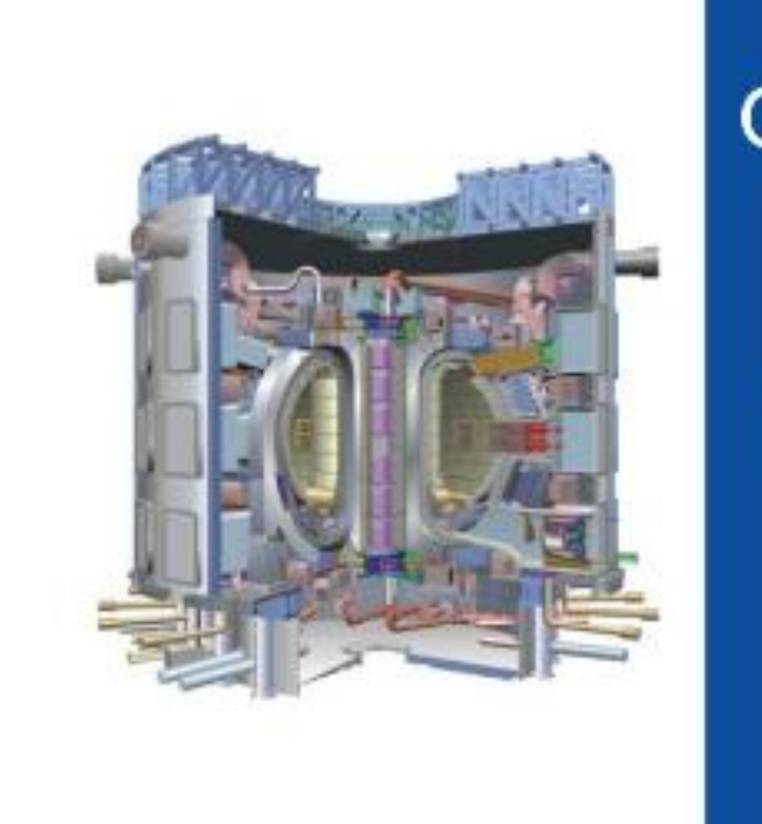
"Our team has been evaluating OpenACC as a pathway to performance portability for the Model for Prediction (MPAS) atmospheric model. Using this approach on the MPAS dynamical core, we have achieved performance on a single P100 GPU equivalent to 2.7 dual socketed Intel Xeon nodes on our new Cheyenne supercomputer."



VMD

John Stone
Senior Research Programmer
Beckman Institute
University of Illinois

"Due to Amdahl's law, we need to port more parts of our code to the GPU if we're going to speed it up. But the sheer number of routines poses a challenge. OpenACC directives give us a low-cost approach to getting at least some speed-up out of these second-tier routines. In many cases it's completely sufficient because with the current algorithms, GPU performance is bandwidth-bound."



GTC

Zhihong Lin
Professor and Principal Investigator
UC Irvine

"Using OpenACC our scientists were able to achieve the acceleration needed for integrated fusion simulation with a minimum investment of time and effort in learning to program GPUs."

OpenACC

More Science, Less Programming

Map courtesy University of Tokyo

GAMERA

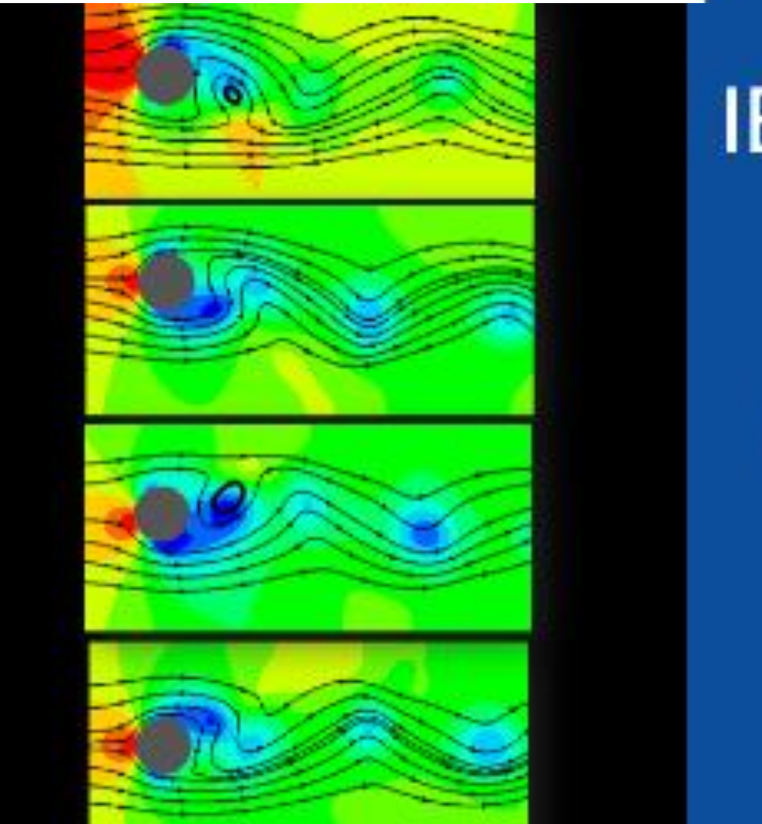
Takuma Yamaguchi, Kohei Fujita, Tsuyoshi Ichimura, Muneo Hori, Lalith Wijerathne
The University of Tokyo

"With OpenACC and a compute node based on NVIDIA's Tesla P100 GPU, we achieved more than a 14X speed up over a K Computer node running our earthquake disaster simulation code"

SANJEEVINI

Abhilash Jayaraj
Project Scientist
Indian Institute of Technology
New Delhi

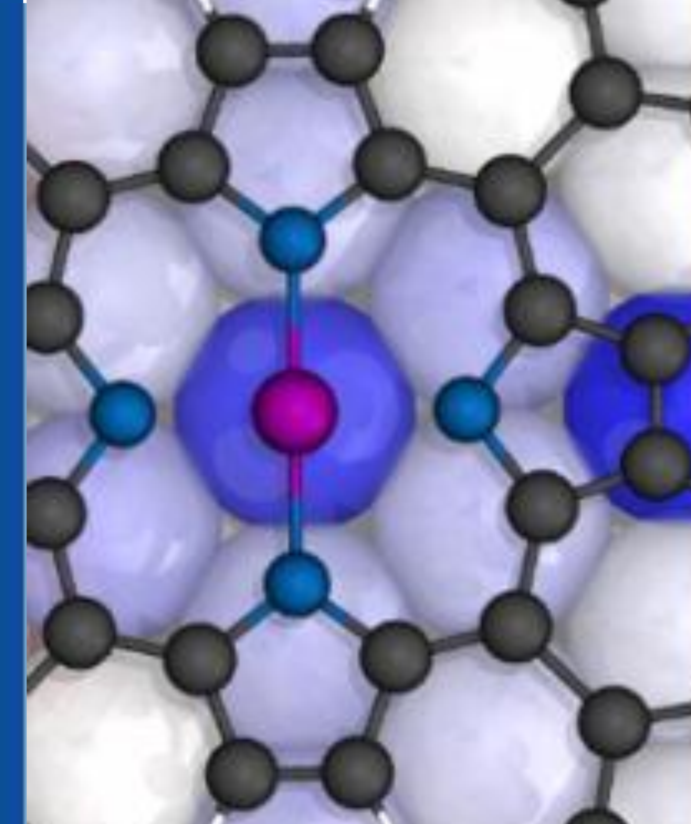
"In an academic environment maintenance and speedup of existing codes is a tedious task. OpenACC provides a great platform for computational scientists to accomplish both tasks without involving a lot of efforts or manpower in speeding up the entire computational task."



IBM-CFD

Somnath Roy
Assistant Professor
Mechanical Engineering Department
Indian Institute of Technology Kharagpur

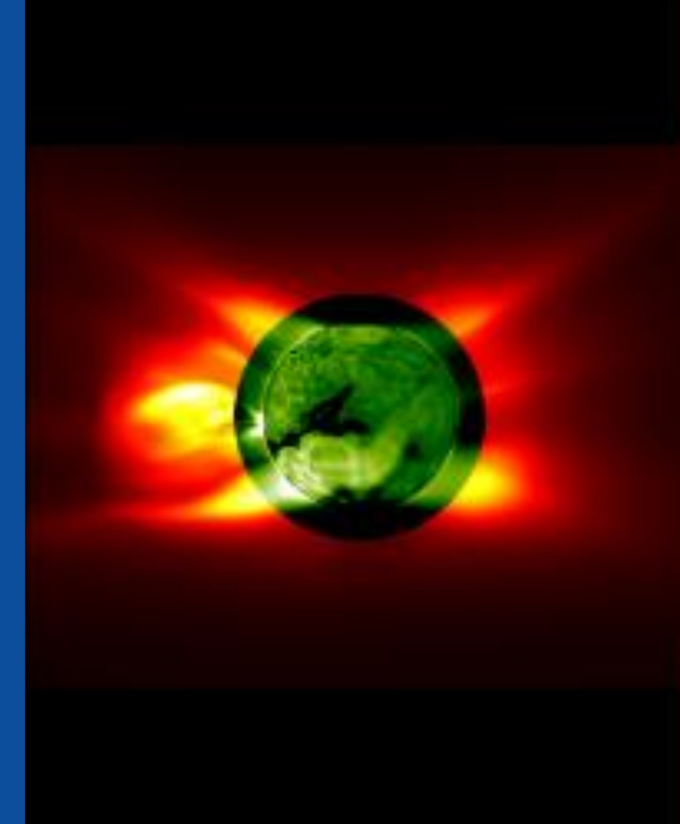
"OpenACC can prove to be a handy tool for computational engineers and researchers to obtain fast solution of non-linear dynamics problem. In immersed boundary incompressible CFD, we have obtained order of magnitude reduction in computing time by porting several components of our legacy codes to GPU. Especially the routines involving search algorithm and matrix solvers have been well-accelerated to improve the overall scalability of the code."



PWscf (Quantum ESPRESSO)

Filippo Spiga
Senior Contributor
Quantum ESPRESSO group

"CUDA Fortran gives us the full performance potential of the CUDA programming model and NVIDIA GPUs. While leveraging the potential of explicit data movement, !$CUF KERNELS directives give us productivity and source code maintainability. It's the best of both worlds."



MAS

Ronald M. Caplan
Computational Scientist
Predictive Science Inc.

"Adding OpenACC into MAS has given us the ability to migrate medium-sized simulations from a multi node CPU cluster to a single multi-GPU server. The implementation yielded a portable single-source code for both CPU and GPU runs. Future work will add OpenACC to the remaining model features, enabling GPU-accelerated realistic solar storm modeling."

OpenACC Directives

Manage Data Movement

Initiate Parallel Execution

Optimize Loop Mappings

```
#pragma acc data copyin(a,b) copyout(c)
{
  ...
  #pragma acc parallel
  {
  #pragma acc loop gang vector
      for (i = 0; i < n; ++i) {
          c[i] = a[i] + b[i];
          ...
      }
  }
}
```

- Incremental
- Single source
- Interoperable
- Performance portable
- CPU, GPU, Manycore

OpenACC

Directives for Accelerators

NVIDIA.

OpenACC Syntax

Syntax for using OpenACC directives in code

C/C++

```
#pragma acc directive clauses
<code>
```

Fortran

```
!$acc directive clauses
<code>
```

A ***pragma*** in C/C++ gives instructions to the compiler on how to compile the code. Compilers that do not understand a particular pragma can freely ignore it.

A ***directive*** in Fortran is a specially formatted comment that likewise instructions the compiler in its compilation of the code and can be freely ignored.

"***acc***" informs the compiler that what will come is an OpenACC directive

Directives are OpenACC commands for altering our code.

Clauses are specifiers or additions to directives.

OpenACC Parallel Directive

Expressing parallelism

```
#pragma acc parallel
{
```

When encountering the ***parallel*** directive, the compiler will generate *1 or more parallel* ***gangs***, which execute redundantly.

```
}
```

OpenACC Parallel Directive

Expressing parallelism

```
#pragma acc parallel
{
    for(int i = 0; i < N; i++)
    {
        // Do Something
    }
}
```

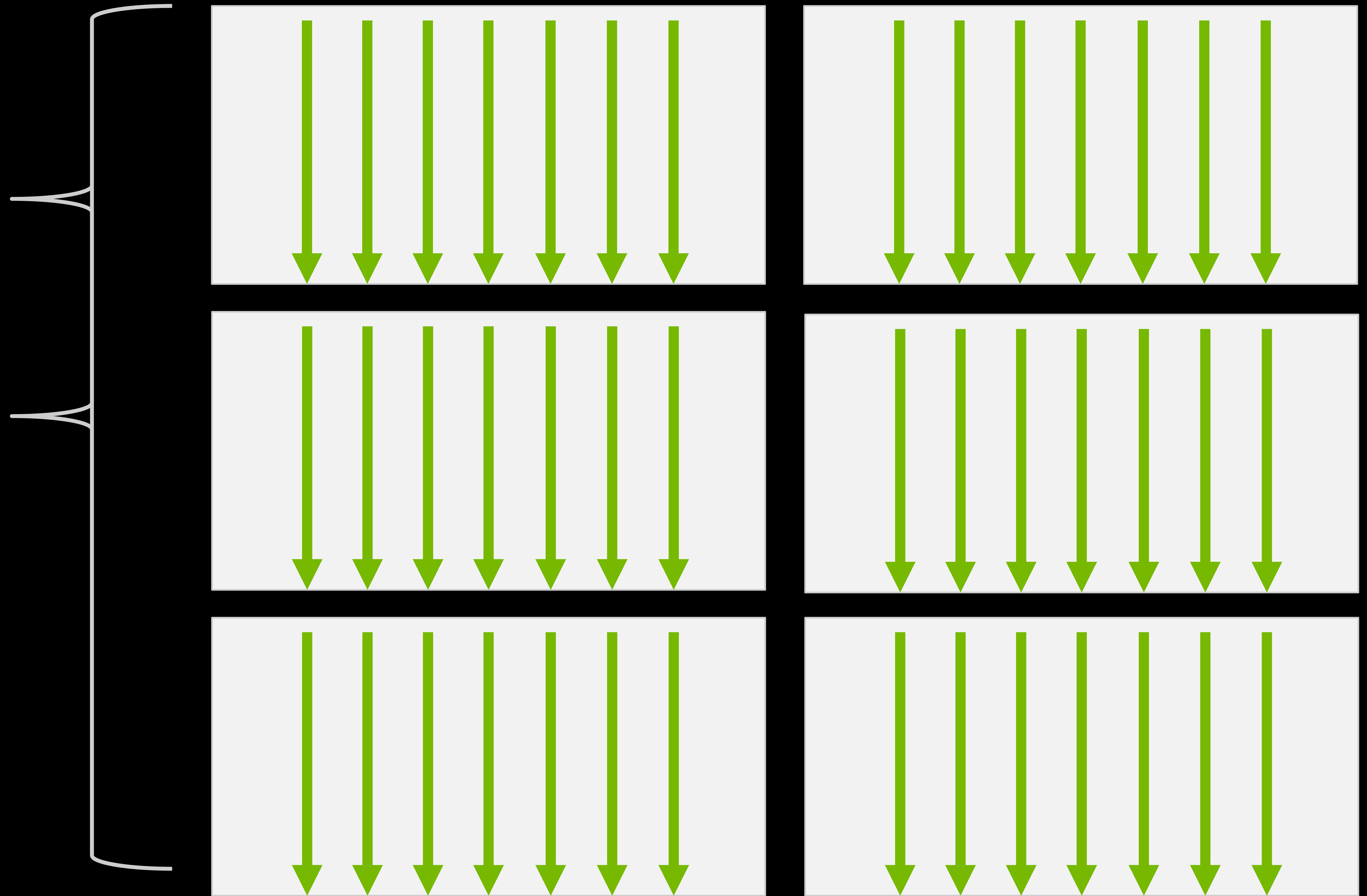
loop

This loop will be executed redundantly on each gang

NVIDIA

```
#pragma acc parallel
{

    #pragma acc loop
    for(int i = 0; i < N; i++)
    {
        // Do Something
    }

}
```

The *loop* directive informs the compiler which loops to parallelize.



NVIDIA

OpenACC Parallel Directive

Parallelizing a single loop

C/C++

```
#pragma acc parallel
{
  #pragma acc loop
  for(int i = 0; j < N; i++)
    a[i] = 0;
}
```

Fortran

```
!$acc parallel
       !$acc loop
       do i = 1, N
              a(i) = 0
       end do
!$acc end parallel
```

Use a **parallel** directive to mark a region of code where you want parallel execution to occur

This parallel region is marked by curly braces in C/C++ or a start and end directive in Fortran

The **loop** directive is used to instruct the compiler to parallelize the iterations of the next loop to run across the parallel gangs

NVIDIA.

OpenACC Parallel Directive

Parallelizing a single loop

C/C++

```
#pragma acc parallel loop
for(int i = 0; j < N; i++)
        a[i] = 0;
```

Fortran

```
!$acc parallel loop
do i = 1, N
        a(i) = 0
end do
```

This pattern is so common that you can do all of this in a single line of code

In this example, the parallel loop directive applies to the next loop

This directive both marks the region for parallel execution and distributes the iterations of the loop.

When applied to a loop with a data dependency, parallel loop may produce incorrect results

Building the code

nvc –fast –Minfo=accel –ta=tesla:managed main.c

nvfortran –fast –Minfo=accel –ta=tesla:managed main.f90

-Minfo shows more details

```
$ nvc -fast -ta=multicore -Minfo=accel laplace2d_uvm.c
main:
     63, Generating Multicore code
         64, #pragma acc loop gang
     64, Accelerator restriction: size of the GPU copy of Anew,A is unknown
         Generating reduction(max:error)
     66, Loop is parallelizable
```

```
$ nvc -fast -ta=tesla:managed -Minfo=accel rdf.c
main:
     63, Accelerator kernel generated
         Generating Tesla code
         64, #pragma acc loop gang /* blockIdx.x */
             Generating reduction(max:error)
         66, #pragma acc loop vector(128) /* threadIdx.x */
     63, Generating implicit copyin(A[:])
Generating implicit copy(error)
     66, Loop is parallelizable
```

NVIDIA

Radial Distribution Function

Pseudo Code

```
for (frame=0;frame<nconf;frame++) {
    for (int id1=0;id1<numatm;id1++)  {
        for (id2=0;id2<numatm;id2++)  {
            dx=d_x[]-d_x[];
            dy=d_y[]-d_y[];
            dz=d_z[]-d_z[];
            r=sqrtf(dx*dx+dy*dy+dz*dz);

            if (r<cut) {
                ig2 = (int)(r/del);
                d_g2[ig2] = d_g2[ig2] +1 ;
            }
        }
    }
}
```

- Across Frames
- Find Distance
- Reduction

Radial Distribution Function

Pseudo Code - C

```
#pragma acc parallel loop
  for (frame=0;frame<nconf;frame++) {
      for (int id1=0;id1<numatm;id1++) {
          for (id2=0;id2<numatm;id2++) {
              dx=d_x[]-d_x[];
              dy=d_y[]-d_y[];
              dz=d_z[]-d_z[];
              r=sqrtf(dx*dx+dy*dy+dz*dz);

              if (r<cut) {
                  ig2 = (int)(r/del);
                  #pragma acc atomic
                  d_g2[ig2] = d_g2[ig2] +1 ;
              }
          }
      }
  }
```

- Parallel Loop construct
- Atomic Construct

Radial Distribution Function

Pseudo Code - Fortran

```
do iconf=1,nframes
        if (mod(iconf,1).eq.0) print*,iconf
        !$acc parallel loop
        do i=1,natoms
           do j=1,natoms
              dx=x(iconf,i)-x(iconf,j)
              dy=y(iconf,i)-y(iconf,j)
              dz=z(iconf,i)-z(iconf,j)
               ...
              if(r<cut)then
                 !$acc atomic
                 g(ind)=g(ind)+1.0d0
              endif
           enddo
        enddo
     enddo
```

- Parallel Loop construct
- Atomic Construct

Programming: CUDA

Hello, World! with Device Code

```
__global__ void kernel( void ) {
    printf("Hello from the GPU!");
}


int main( void ) {
    kernel<<<1,1>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

```
module printgpu
contains
  attributes(global) subroutine print_from_gpu()
    implicit none
    print *, "Hello from the GPU!"
  end subroutine print_from_gpu
end module printgpu

program testPrint
  use printgpu
  use cudafor
  implicit none
  integer istat

  call print_from_gpu<<<1, 1>>>()
  istat = cudaDeviceSynchronize();

end program testPrint
```

NVIDIA.

A (slightly!) more complex example

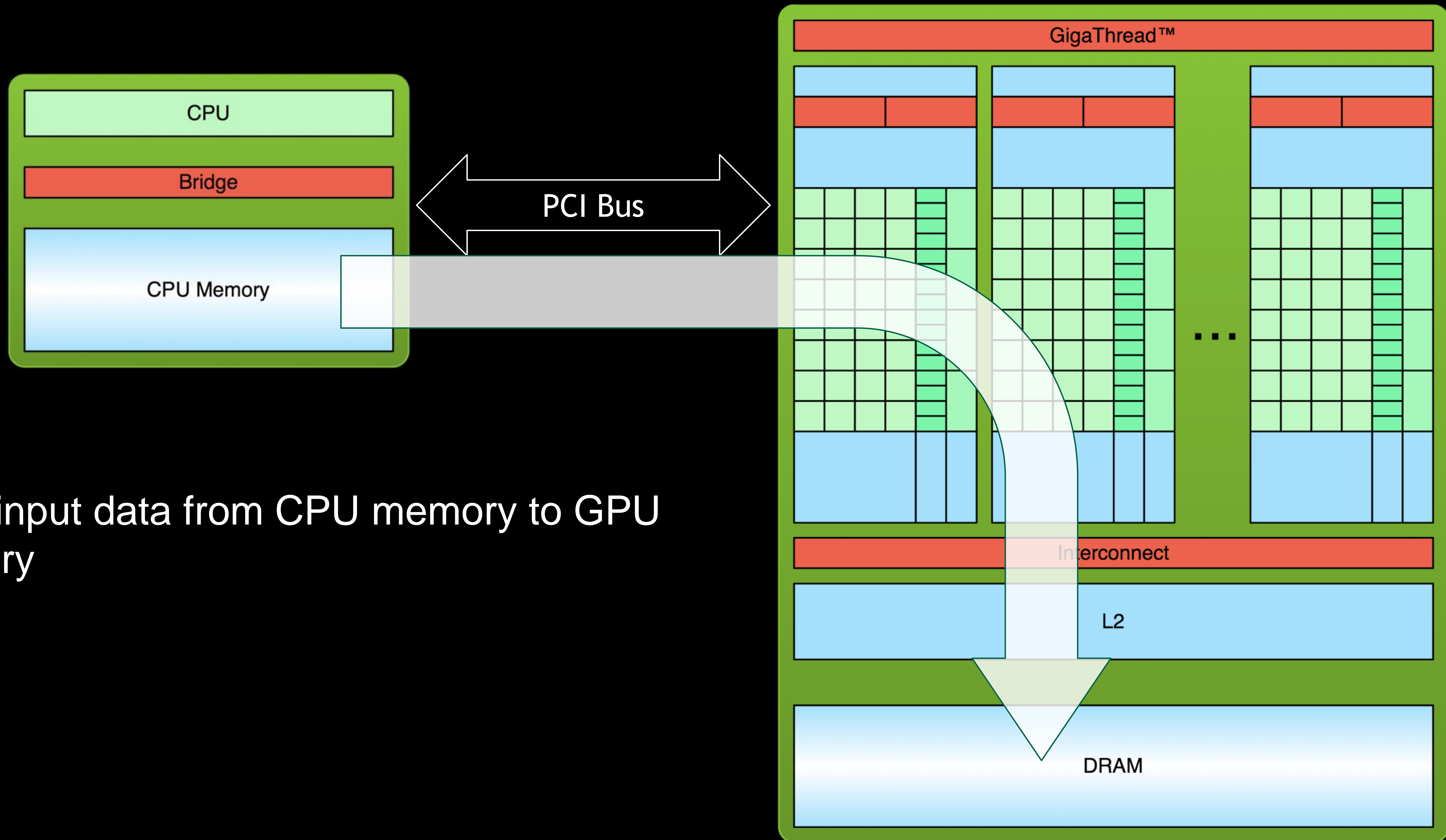
- A simple kernel to add two vectors

```
__global__ void addVector( int *a,
                           int *b, int *c ) {
    . . .
    c[myID] = a[myID] + b[myID];
}
```

```
attributes(global) subroutine addVector( a, b, c)
        int, device :: a(:), b(:), c(:)
        . . .
        c(myID) = a(myID) + b(myID)

end subroutine addVector
```

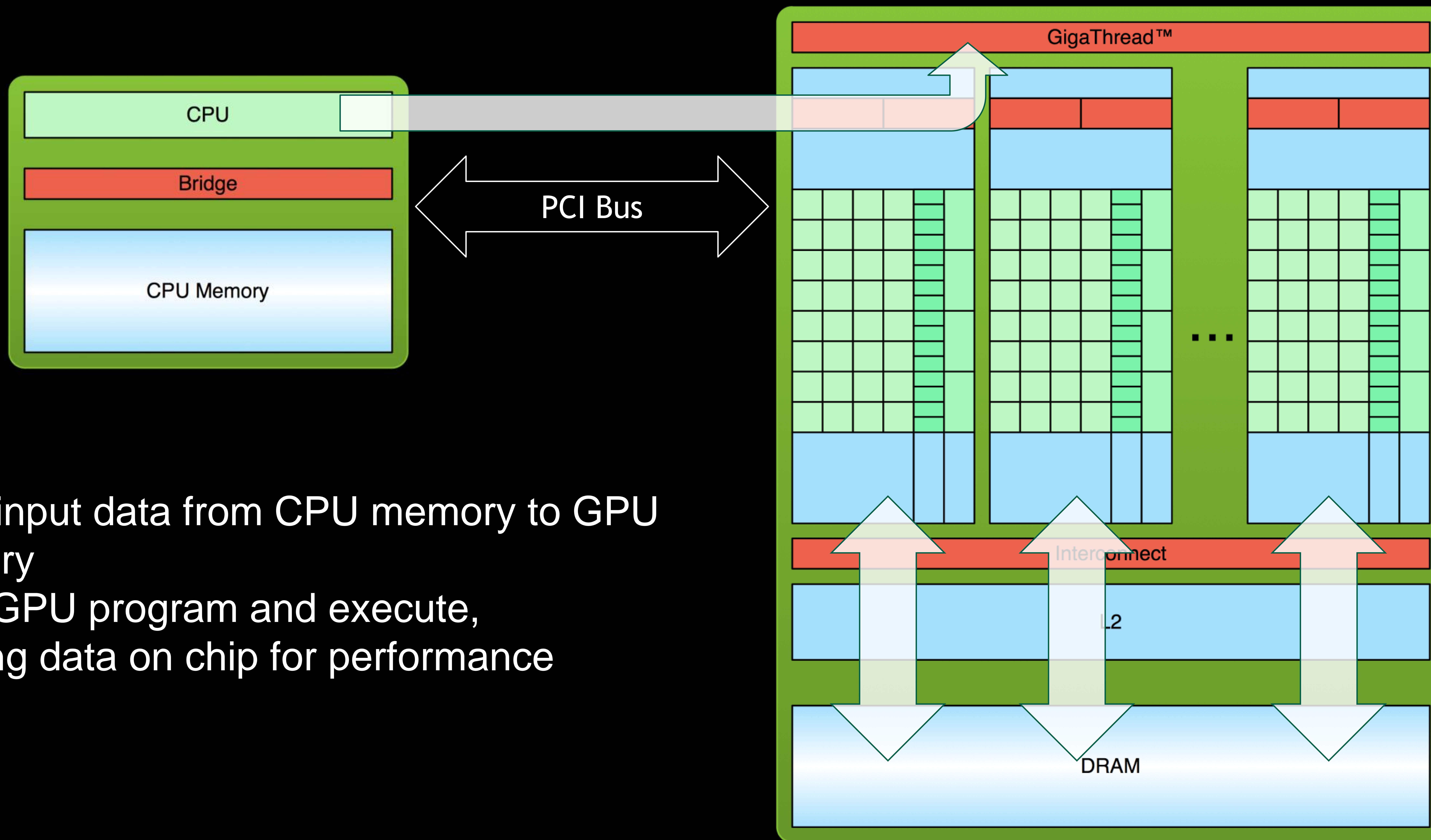
- `addVector()` will execute on the device … so a, b, and c must point to device memory
- How do we allocate memory on the GPU?
- How do we move memory between the CPU and the GPU?

Processing Flow – Step 1



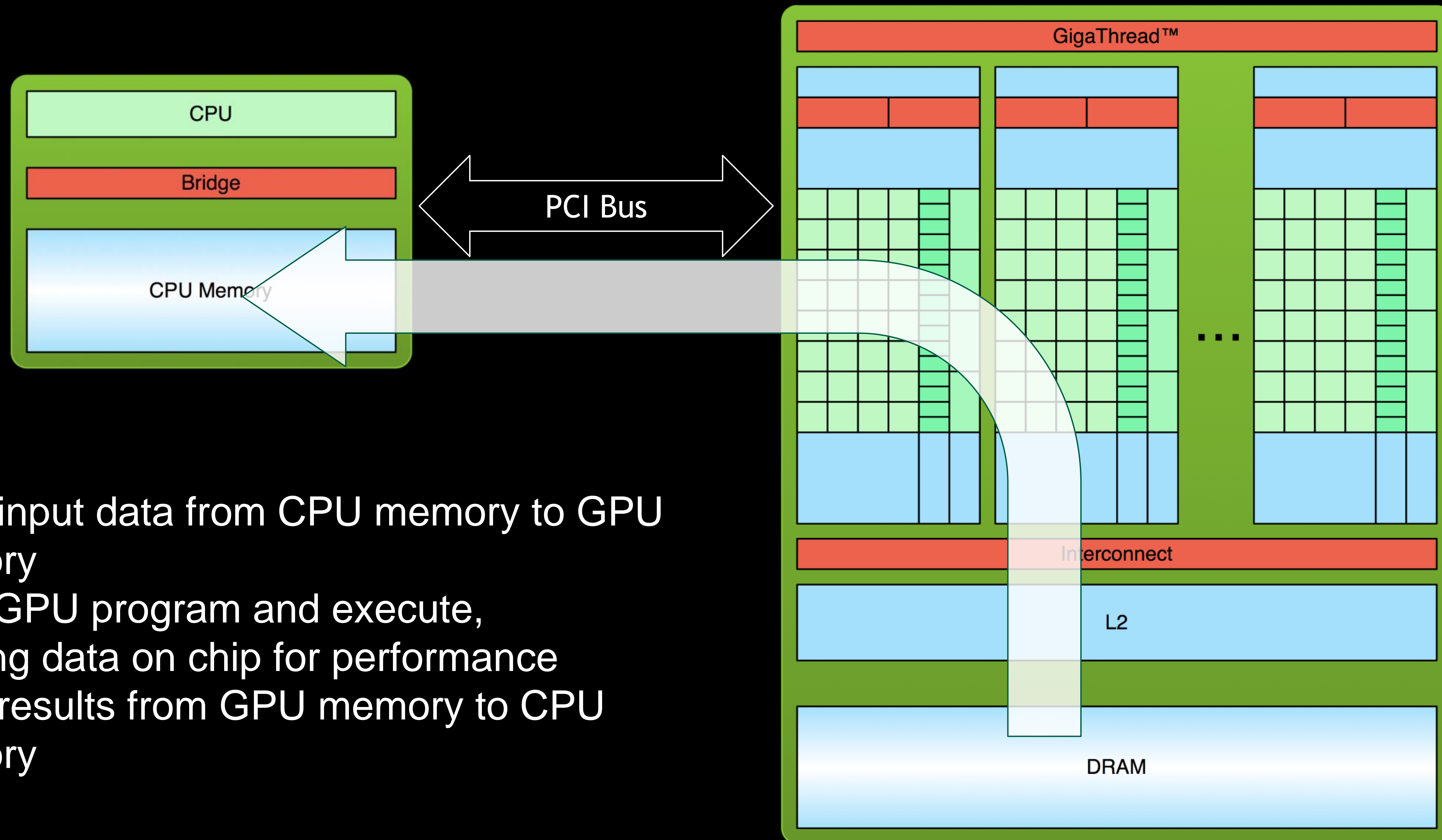
1. Copy input data from CPU memory to GPU memory

Processing Flow – step 2



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Processing Flow – Step 3



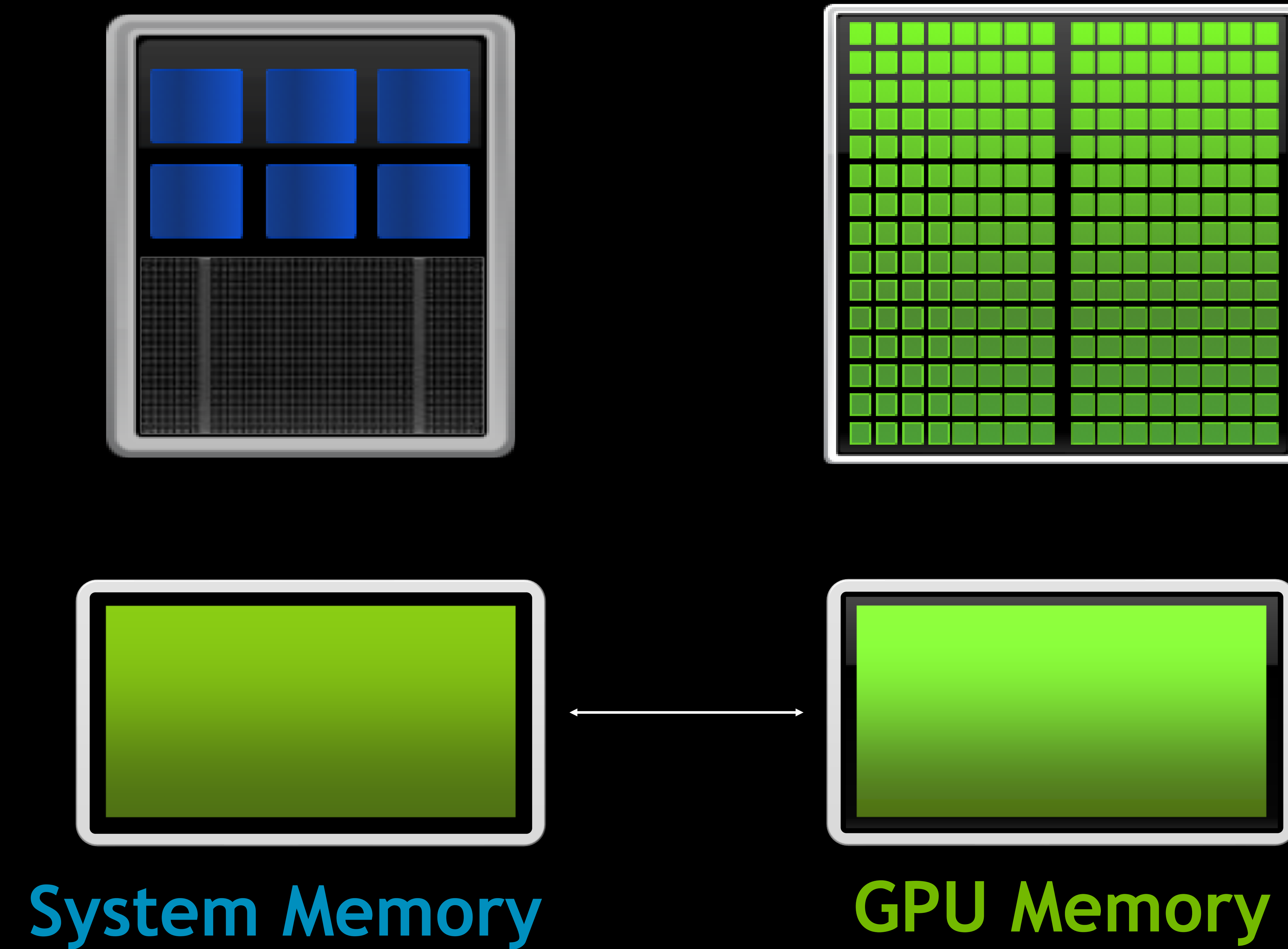
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

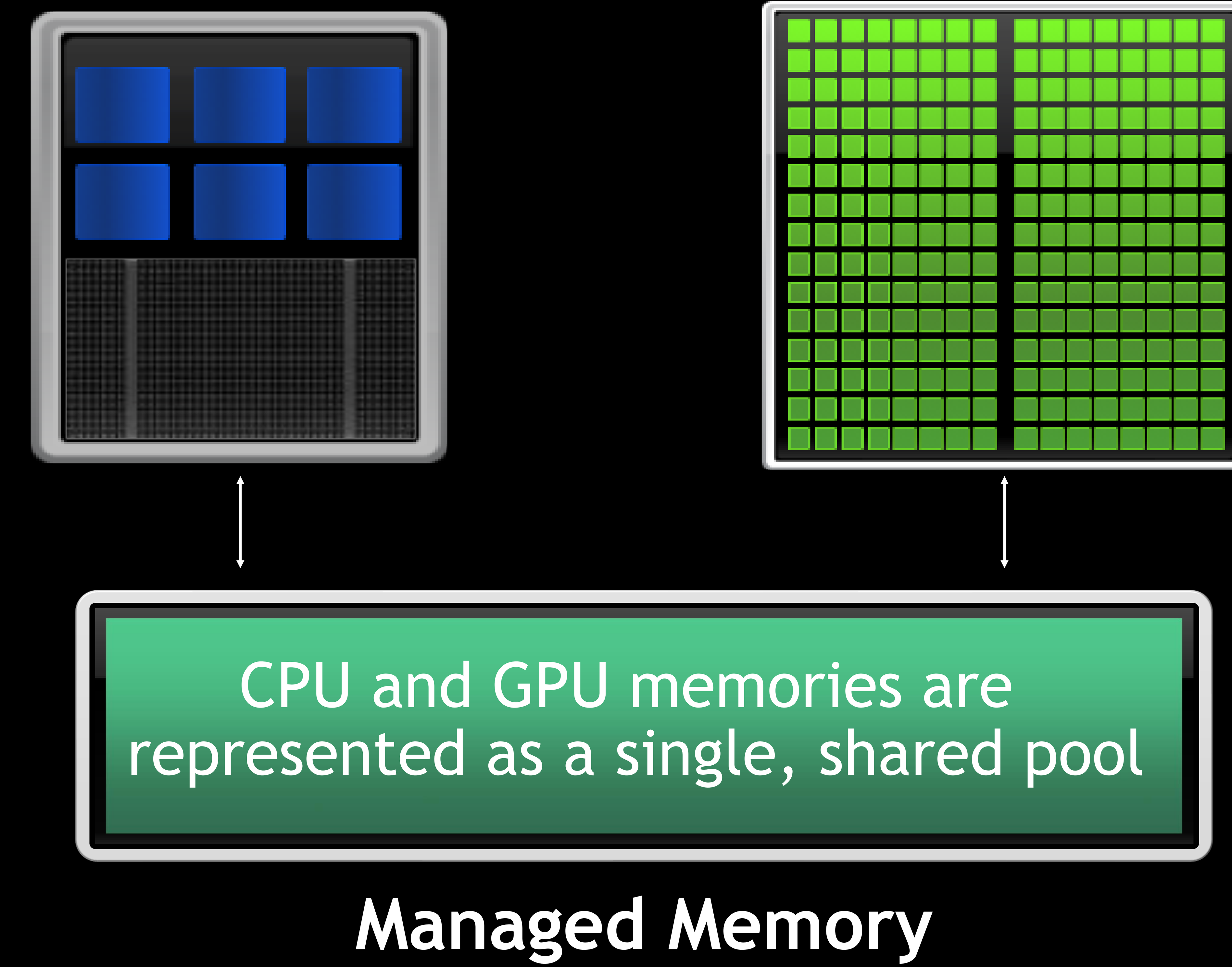
NVIDIA

CUDA Unified Memory

Simplified Developer Effort

Commonly referred to as *"managed memory."*





Memory Management

- Host and device memory are distinct entities

- Basic CUDA API for dealing with explicit device memory management
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to their C equivalents, `malloc()`, `free()`, `memcpy()`
  - Similar to their Fortran equivalents, `allocate()`, `deallocate`
    - Array should be defined as allocatable

- CUDA API for using Unified memory is
  - C API: `cudaMallocManaged()`, `cudaFree()`
  - Fortran: declare variable with managed, allocatable attribute
    - `real, managed, allocatable, dimension(:,:) :: A, B, C`

NVIDIA.

Expressing Parallelism

Parallel Programming in CUDA

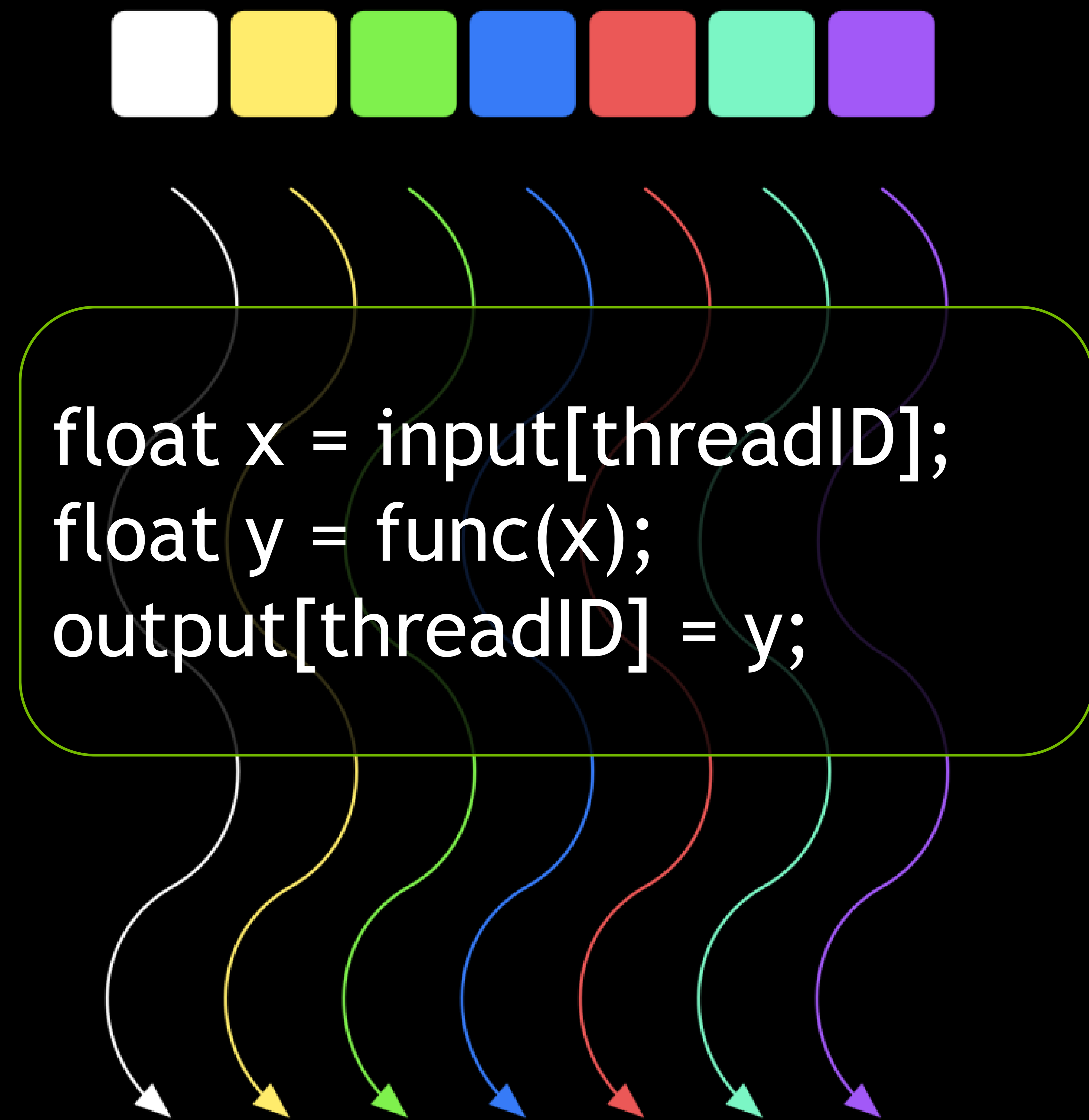
- But wait…GPU computing is about massive parallelism
- So how do we run code in parallel on the device?
- Solution lies in the parameters between the triple angle brackets:

```
doSomethingOnce<<< 1, 1 >>>();

doSomethingLots<<< NumBlocks, NumThreads >>>();
```

- Instead of executing once, doSomethingLots() executes NumBlocks * NumThreads times, in parallel

CUDA Kernels: Parallel Threads

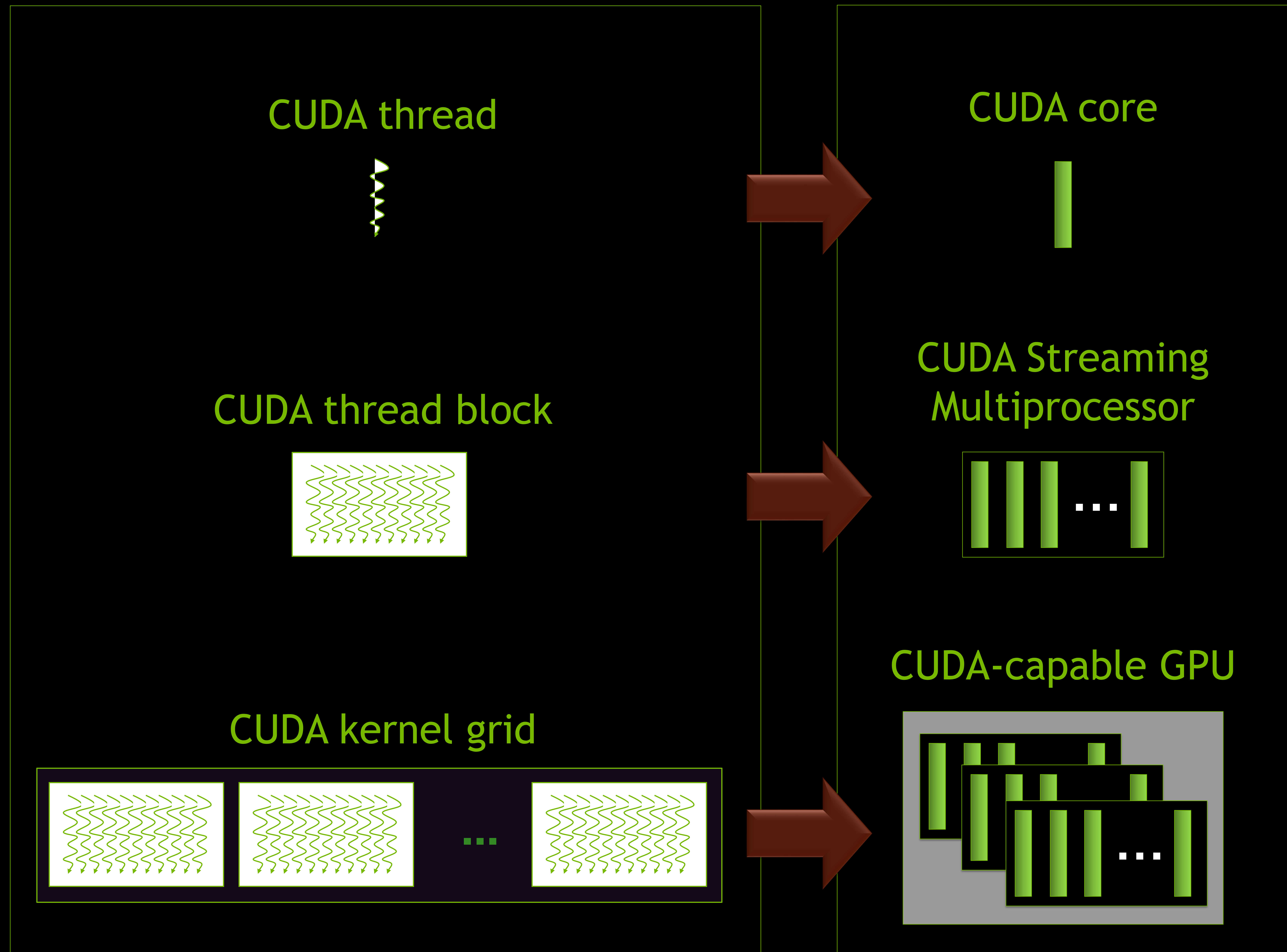
- A kernel is a function executed on the GPU
  - Array of threads, in parallel
- All threads execute the same code, can take different paths
  - Each thread has an ID
  - Select input/output data
  - Control decisions



CUDA Kernels: Subdivide into Blocks

- Instructions issued to 32 threads at once – a warp
- Threads are grouped into blocks – blocks execute on a streaming processor
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

Kernel Execution



- Each thread is executed by a core
- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks' memory requirements and the SM's memory resources
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

Communication Within a Block

- Threads may need to cooperate
  - Memory accesses
  - Share results
- Cooperate using shared memory
  - Accessible by all threads within a block
- Restriction to “within a block” permits scalability
  - Fast communication between N threads is not feasible when N large

General Rules for choice of NUM THREADS/NUM BLOCKS

- Simple example – numThreads*numBlocks chosen to match dataset size
  - Grid-stride loops are an alternative approach which allows us to work with arbitrary sizes

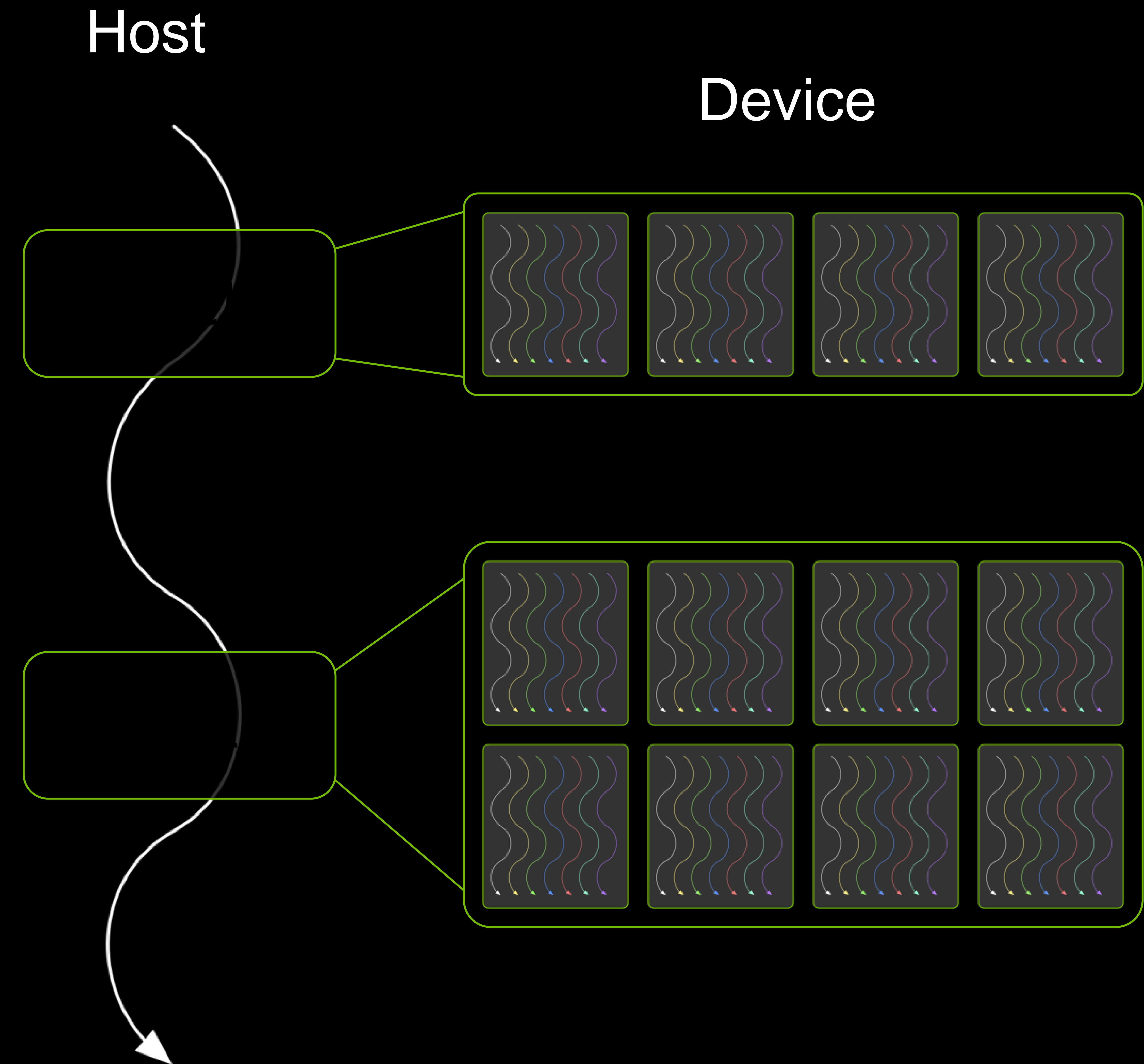
- numThreads = 32*k, k=1, or 2, … or 32
  - Instructions can be issued to 32 threads at once (a warp)
  - Device-specific, but max number of threads *per block* is usually 1024 (i.e. 32 x 32) link

- numBlocks = numSM*m, m=1, 2, …
  - numSM = number of streaming multiprocessors on the device (cudaGetDeviceProperties)

- On V100 <<<80, 64>>> would create same number of threads as there are cores
  - But probably want more threads than this … <<<16*80, 256>>> ?
  - This gives the GPU scheduler opportunity for increased concurrency – latency hiding

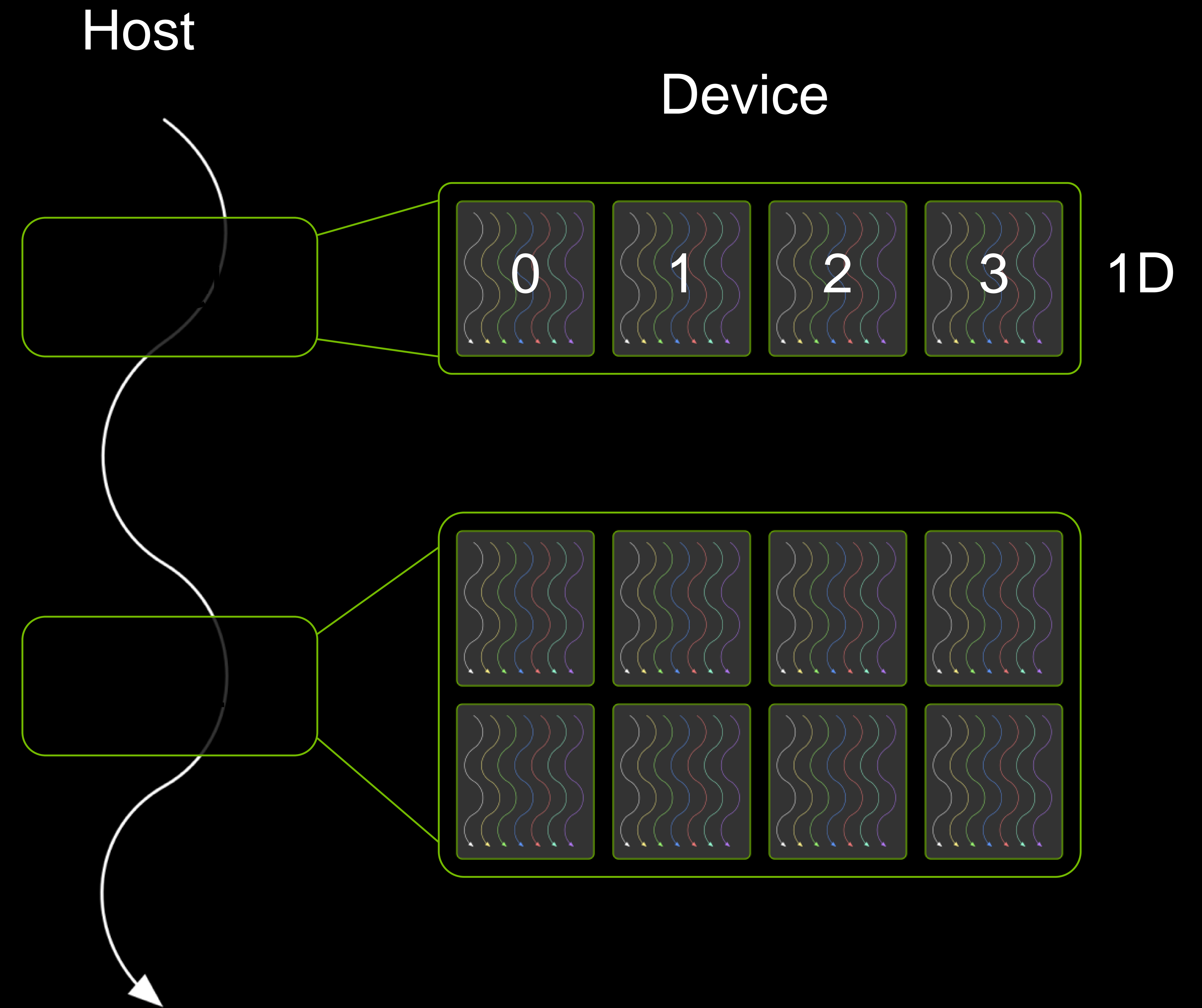
CUDA Programming Model - Summary

- A kernel executes as a grid of thread blocks
- A block is a batch of threads
  - Communicate through shared memory
- Each block has a block ID
- Each thread has a thread ID



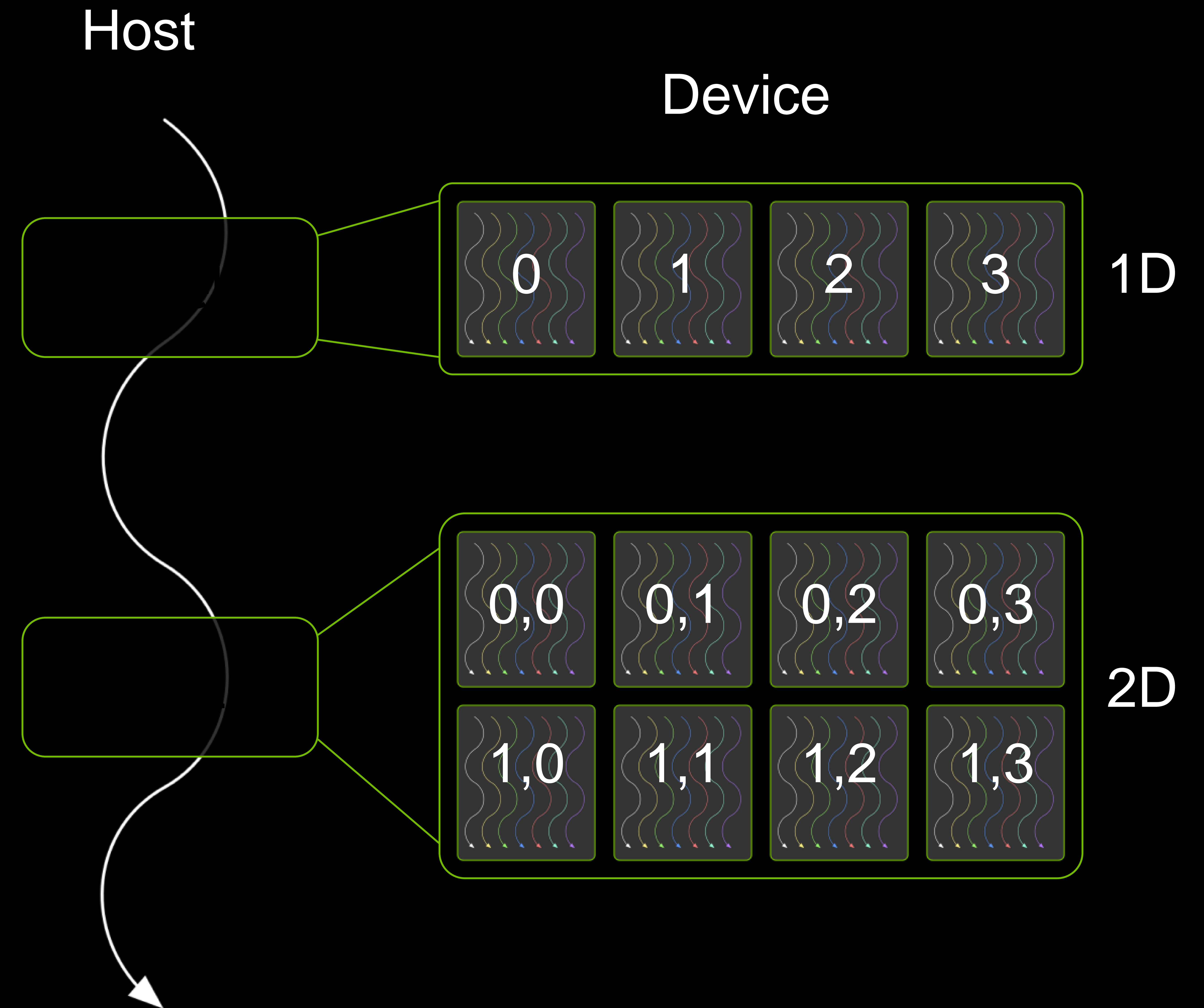
CUDA Programming Model - Summary

- A kernel executes as a grid of thread blocks
- A block is a batch of threads
  - Communicate through shared memory
- Each block has a block ID
- Each thread has a thread ID



CUDA Programming Model - Summary

- A kernel executes as a grid of thread blocks
- A block is a batch of threads
  - Communicate through shared memory
- Each block has a block ID
- Each thread has a thread ID



NVIDIA

Back to our addVector kernel

Our vectors have N elements …

```
__global__ void addVector( int *a,
                     int *b, int *c ) {
     …
     c[myID] = a[myId] + b[myID];
}

int main( void ) {
    …
    numThreads = 32;
    numBlocks  = N/numThreads; // Assumption
    addVector<<<numBlocks,numThreads>>>(a, b,
c);
    cudaDeviceSynchronize();
    return 0;
}
```

```
attributes(global) subroutine addVector( a, b, c)
        int, device :: a(:), b(:), c(:)
        …
        c(myID) = a(myID) + b(myID)

end subroutine addVector

program demo
  use cudafor
  implicit none
  integer istat
  …
  numThreads = 32
  numBlocks = N/numThreads ! Assumes N is divisible by 32
  call addVector<<< numBlocks, numThreads >>>(a, b, c)
  istat = cudaDeviceSynchronize();

end program demo
```

BUT what about the kernel itself?

- We now have our kernel executing N times …

```
__global__ void addVector( int *a, int *b,
                           int *c ) {
        …
        c[myID] = a[myId] + b[myID];
}
```

```
attributes(global) subroutine addVector( a, b, c)
        int, device :: a(:), b(:), c(:)
        …
        c(myID) = a(myID) + b(myID)

end subroutine addVector
```

- We want *each* thread in our grid of threads to operate on *one* data element in our vector
- But how do we determine which thread, in which block, is currently executing?

CUDA-DEFINED VARIABLES FOR iNDEXING

- blockDim.x – number of threads per block
- threadIdx.x – index of a thread within a block: 0 .. blockDim.x-1 (FORTRAN 1..)
- gridDim.x – number of blocks in the grid
- blockIdx.x – index of a block within the grid: 0 .. gridDim.x-1

- globalThreadID = threadIdx.x + blockIdx.x * blockDim.x

- Similarly for Fortran
- globalThreadID = threadidx%x + (blockidx%x-1)*blockdim%x

Back to our addVector example

```
__global__ void addVector( int *a, int *b, int
*c ) {
    …
    myID = threadIdx.x+blockIdx.x*blockDim.x;
    c[myID] = a[myId] + b[myID];
}
```

```
attributes(global) subroutine addVector( a, b, c)
        int, device :: a(:), b(:), c(:)
        …
        myID = threadidx%x + (blockidx%x-1)*blockdim%x
        c(myID) = a(myID) + b(myID)

end subroutine addVector
```

One last thing ...

One last thing ...

Programming Model Interoperability

All programming models are interoperable:

- Can be used in the same source file.
- ABI compatible with each other and NVCC.

```
nvc++ -stdpar -cuda -acc -mp
```

CUDA

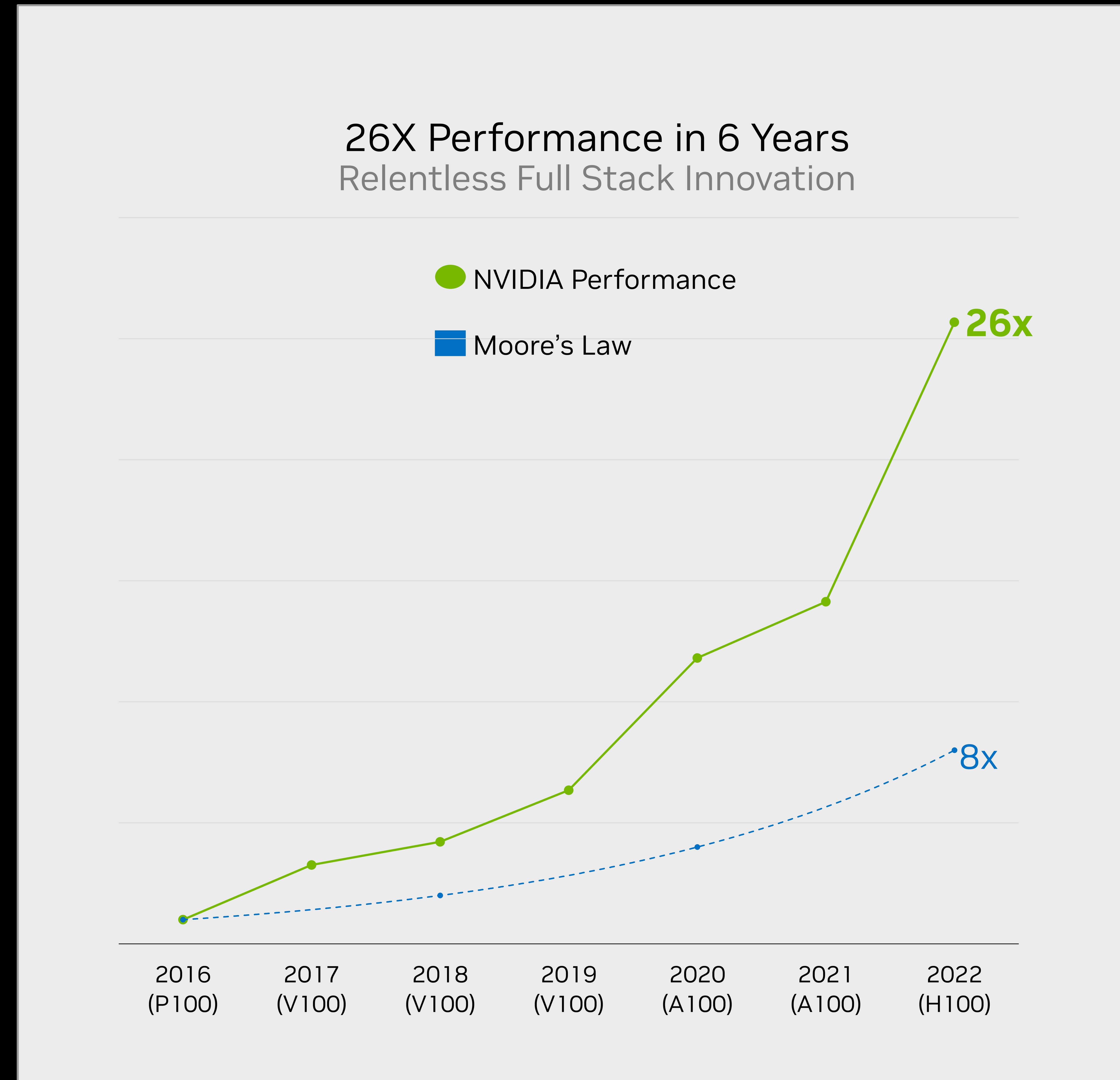
OpenACC

OpenMP

```
__global__ void
times_two(float* first, uint64_t n) {
  auto t = blockIdx.x*blockDim.x+threadIdx.x;
  if (t < n) first[t] *= 2;
}

void compute(std::vector<float>& x) {
  auto const b = ((x.size() - 1)/ 64) + 1;
  times_two<<<b, 64>>>(x.data(), x.size());
  cudaDeviceSynchronize();

  #pragma omp target teams loop
  for (auto& e : x) e *= e;

  #pragma acc parallel loop
  for (auto& e : x) e += e;

  std::sort(std::execution::par,
            x.begin(), x.end());
}
```

Final thought

Keep APPS, LIBRARIES and FRAMEWORKS up to date



Center Panel: Geometric mean of application speedups vs. P100 | benchmark applications | Amber [PME-Cellulose NVE], Chroma [HMC], GROMACS [ADH Dodec], MILC [Apex Medium], NAMD [stmv_nve_cuda], PyTorch (BERT Large Fine Tuner], Quantum Espresso [AUSURF112-jR]; TensorFlow [ResNet-50], VASP 6 [Si Huge], |GPU node: with dual-socket CPUs with 4x P100, V100, or A100 GPUs. H100 values shown for 2022 projected performance subject to change

NVIDIA PAX-HPC Workshop
17th January 2024

Thankyou!

Paul Graham | Senior Solutions Architect

pgraham@nvidia.com